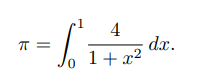
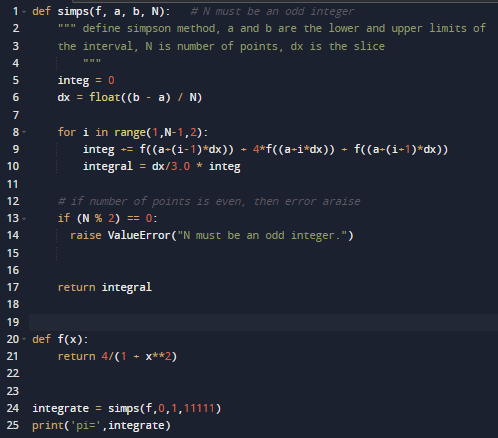
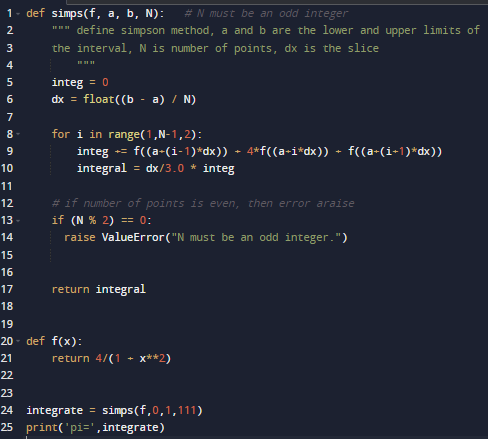
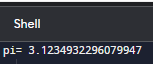
***Homework 4***

*Ex. 4.2*

*Simpson’s Rule*

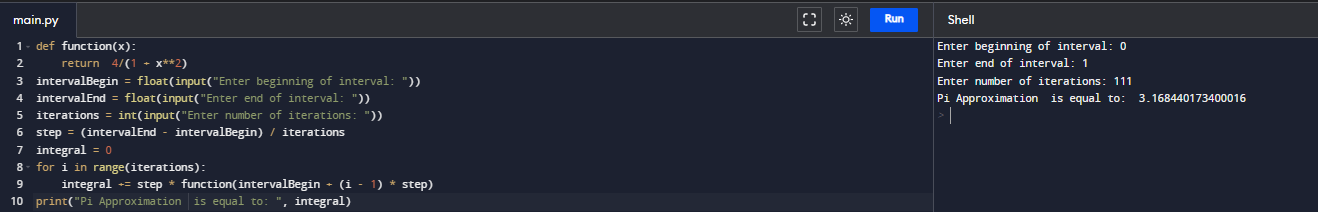
**

N=11111

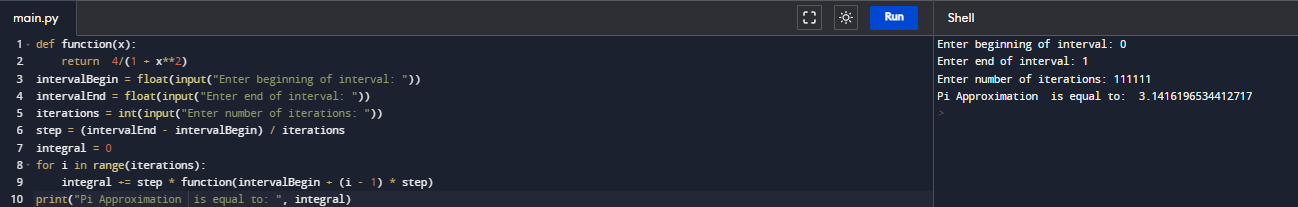
N=111

*Midpoint Method*

*(I’m variating the stepsize by different numbers of iterations, so then the number of iterations is greater, the approximation is more accurate)*

**

N=111

**

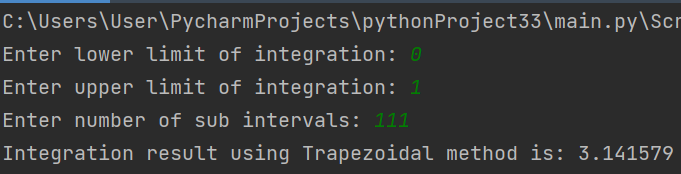
N=111111

*Trapezoidal method*

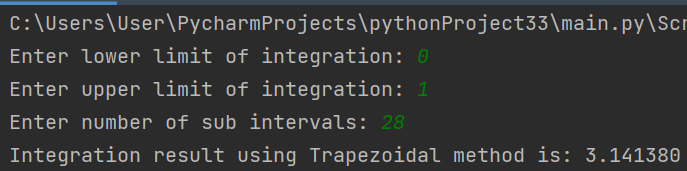
# Trapezoidal Method  
  
def f(x):  
 return 4 / (1 + x \*\* 2)  
  
def trapezoidal(x0, xn, n):  
 h = (xn - x0) / n  
   
 integration = f(x0) + f(xn)  
  
 for i in range(1, n):  
 k = x0 + i \* h  
 integration = integration + 2 \* f(k)  
  
 integration = integration \* h / 2  
  
 return integration  
lower\_limit = float(input("Enter lower limit of integration: "))  
upper\_limit = float(input("Enter upper limit of integration: "))  
sub\_interval = int(input("Enter number of sub intervals: "))  
  
result = trapezoidal(lower\_limit, upper\_limit, sub\_interval)  
print("Integration result using Trapezoidal method is: %0.6f" % (result))

*Algorithm!*

1. *Define a function to integrate*
2. *Calculating step size*
3. *Implementing trapezoidal method*
4. *Finding the sum*
5. *Finding final integration value*
6. *Input section for limits and subinterval*
7. *Call the trapezoidal() method and get the result*

**

The first case with N=111

**

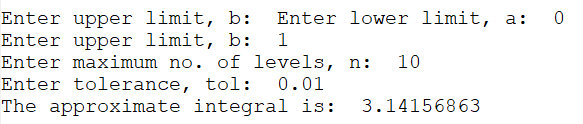
The second case with N=28

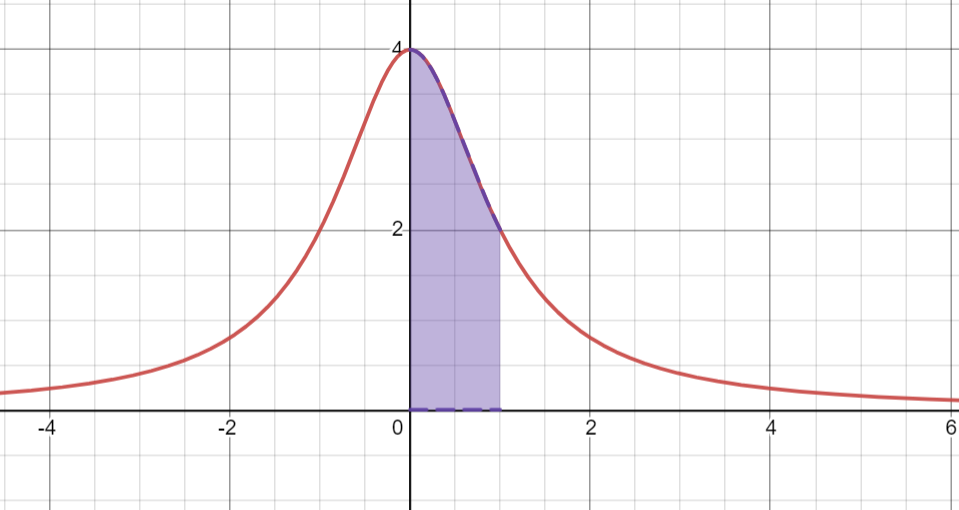
*4.2b*

% Adaptive quadrature algorithm   
 % Find the integral of 4/(1+(x)^2) from 0 to 1.  
  
 f = @(x) 4/(1+(x)^2);  
 aa = input('Enter lower limit, a: ');  
 bb = input('Enter upper limit, b: ');  
 n = input('Enter maximum no. of levels, n: ');  
 eps = input('Enter tolerance, tol: ');  
  
 cnt = 0;  
 app = 0;  
 i = 1;  
 tol = zeros(1,n);  
 a = zeros(1,n);  
 h = zeros(1,n);  
 fa = zeros(1,n);  
 fc = zeros(1,n);  
 fb = zeros(1,n);  
 s = zeros(1,n);  
 l = zeros(1,n);  
 fd = zeros(1,n);  
 fe = zeros(1,n);  
 v = zeros(1,7);  
 tol(i) = 10\*eps;  
 a(i) = aa;  
 h(i) = 0.5\*(bb-aa);  
 fa(i) = f(aa);  
 cnt = cnt+1;  
 fc(i) = f((aa+h(i)));  
 cnt = cnt+1;  
 fb(i) = f(bb);  
 cnt = cnt+1;  
 s(i) = h(i)\*(fa(i)+4\*fc(i)+fb(i))/3;  
 l(i) = 1;  
 while i > 0   
 fd = f((a(i)+0.5\*h(i)));  
 cnt = cnt+1;  
 fe = f((a(i)+1.5\*h(i)));  
 cnt = cnt+1;  
 s1 = h(i)\*(fa(i)+4\*fd+fc(i))/6;  
 s2 = h(i)\*(fc(i)+4\*fe+fb(i))/6;  
 v(1) = a(i);  
 v(2) = fa(i);  
 v(3) = fc(i);  
 v(4) = fb(i);  
 v(5) = h(i);  
 v(6) = tol(i);  
 v(7) = s(i);  
 lev = l(i);  
 i = i-1;  
 if abs(s1+s2-v(7)) < v(6)  
 app = app+(s1+s2);  
 else  
 if lev >= n  
 fprintf('Procedure fails');  
 else  
 i = i+1;  
 a(i) = v(1)+v(5);  
 fa(i) = v(3);  
 fc(i) = fe;  
 fb(i) = v(4);  
 h(i) = 0.5\*v(5);  
 tol(i) = 0.5\*v(6);  
 s(i) = s2;  
 l(i) = lev+1;  
 i = i+1;  
 a(i) = v(1);  
 fa(i) = v(2);  
 fc(i) = fd;  
 fb(i) = v(3);  
 h(i) = h(i-1);  
 tol(i) = tol(i-1);  
 s(i) = s1;  
 l(i) = l(i-1);  
 end  
 end  
 end  
 fprintf('The approximate integral is: %11.8f \n', app);

*Adaptive Quadrature Algorithm using MATLAB (m file).*

*The idea of adaptive quadrature is that if we can approximate each of the two integrals on the right to within a specified tolerance, then the sum gives us the desired result. The result is shown in the windows command*

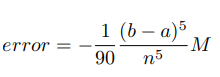
**

**

*Simpson's rule approximations achieve a given level of accuracy faster than trapezoidal. Simpson's Rule is even more accurate than the Trapezoid Rule.*

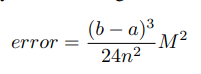
*In fact, the Midpoint can achieve the accuracy of the Simpsons at a very large n. Also, I found that the error in the Trapezoidal is almost twice the error in the Midpoint, but in opposite direction. I learned that Simpson's rule converges the fastest and the most accurate.*

***Error for Simpson’s Rule***

*This method has the following error: *

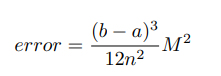
*where M is the maximum of the fourth derivative. this error is the smallest one of these three rules.*

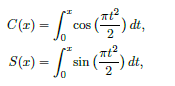
***Error for Midpoint Rule***

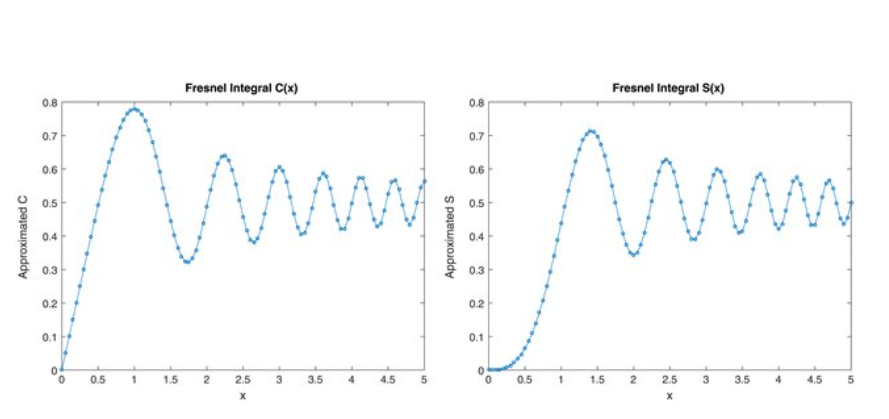
* The midpoint rule is one of the least accurate methods, however, it gives us quite an accurate approximation in the case, when the function doesn’t change a lot in the subintervals.*

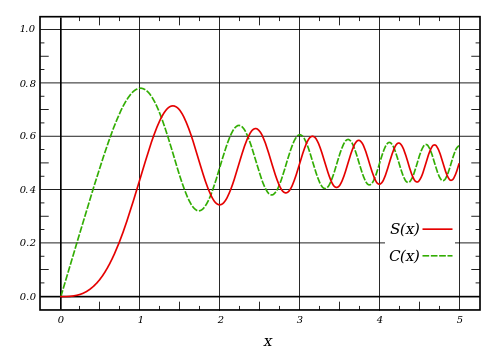
*Where M is the maximum of the second derivative of the function f. According to this formula, we can see, that the significant impact of the value of error has a number n of subintervals – the bigger n, the smaller error.*

***Error for Trapezoid Rule***

* and similar to the midpoint rule, M is the maximum of the second derivative*

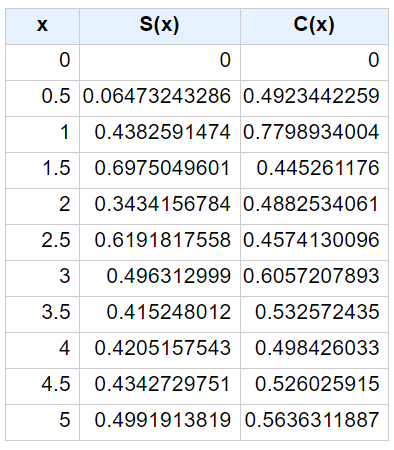
***Ex. 4.3***

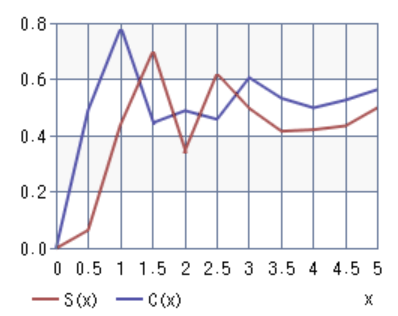
**

**

*Matlab Code*

function f = fresnel(x,N)  
% Evaluates the approximation F\_N(x) to the Fresnel integral F(x).  
% x is a real scalar or matrix,  
% N is the positive integer controlling accuracy (suggest N=12),  
% f is the corresponding scalar or matrix of values of F\_N(x).  
select = x>=0;  
f = zeros(size(x));  
if any(select), f(select) = F(x(select),N); end  
if any(~select), f(~select) = 1-F(-x(~select),N); end  
function f = F(x,N)  
h = sqrt(pi/(N+0.5));  
t = h\*((N:-1:1)-0.5); AN = pi/h;  
t2 = t.\*t; t4 = t2.\*t2; et2 = exp(-t2);  
rooti = exp(i\*pi/4);  
z = rooti\*x; x2 = x.\*x; x4 = x2.\*x2; z2 = i\*x2;  
S = (-et2(1)./(x4+t4(1))).\*(z2+t2(1));  
for n = 2:N  
S = S + (-et2(n)./(x4+t4(n))).\*(z2+t2(n));  
end  
ez = exp((2\*AN\*i\*rooti)\*x);  
f = (i/AN)\*z.\*exp(z2).\*S + ez./(ez+1);

**

**

*Initial value=0*

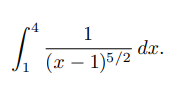
*Increment=0.5*

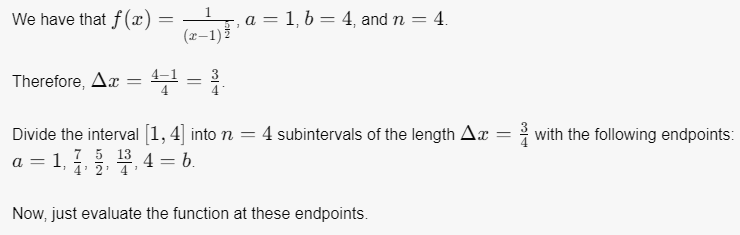
*Repetition=11*

function [C,S] = fresnelCS(x,N)  
% Evaluates approximations to the Fresnel integrals C(x) and S(x).  
% x is a real scalar or matrix,  
% N is a positive integer controlling accuracy (suggest N=12),  
% C and S are the scalars/matrices of the same size as x approximating C(x) and S(x).  
h = sqrt(pi/(N+0.5));  
t = h\*((N:-1:1)-0.5); AN = pi/h; rootpi = sqrt(pi);  
t2 = t.\*t; t4 = t2.\*t2; et2 = exp(-t2);  
x2pi2 = (pi/2)\*x.\*x; x4 = x2pi2.\*x2pi2;  
a = et2(1)./(x4+t4(1)); b = t2(1)\*a;  
for n = 2:N  
term = et2(n)./(x4+t4(n));  
a = a + term; b = b + t2(n)\*term;  
end  
a = a.\*x2pi2;  
mx = (rootpi\*AN)\*x; Mx = (rootpi/AN)\*x;  
Chalf = 0.5\*sign(mx); Shalf = Chalf;  
select = abs(mx)<39;  
if any(select)  
mxs = mx(select); shx = sinh(mxs); sx = sin(mxs);  
den = 0.5./(cos(mxs)+cosh(mxs));  
Chalf(select) = (shx+sx).\*den;  
ssdiff = shx-sx;  
select2 = abs(mxs)<1;  
if any(select2)  
mxs = mxs(select2); mxs3 = mxs.\*mxs.\*mxs; mxs4 = mxs3.\*mxs;  
ssdiff(select2) = mxs3.\*(1/3 + mxs4.\*(1/2520 ...  
+ mxs4.\*((1/19958400)+(0.001/653837184)\*mxs4)));  
end  
Shalf(select) = ssdiff.\*den;  
end  
cx2 = cos(x2pi2); sx2 = sin(x2pi2);  
C = Chalf + Mx.\*(a.\*sx2-b.\*cx2); S = Shalf - Mx.\*(a.\*cx2+b.\*sx2);

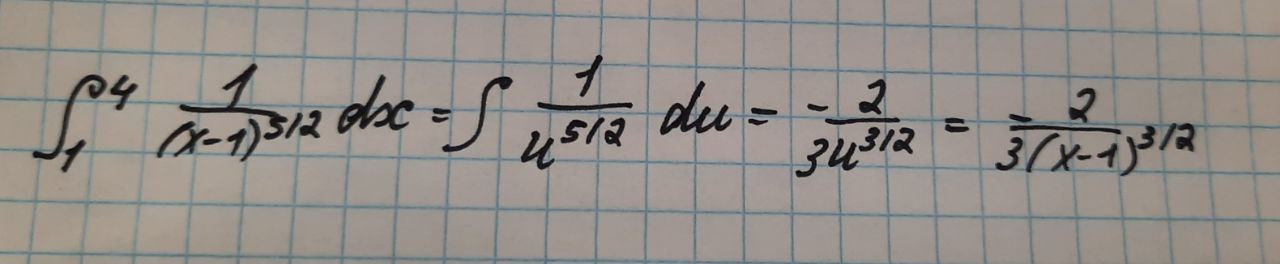
Matlab to evaluate CN (x) and SN (x)

*The code for SN (x) and CN (x) is faster still, but the power series truncated after 15 terms, is more accurate and efficient on the interval [0, 1.5]**An effective computational method for smaller values of |x| is to make use of the power series for C(x) and S(x). These converge for all x, and very rapidly for smaller x, and so are widely used for computation power series for |x| ≤ 1.5. For this range, after the first two terms, these series are alternating series of monotonically decreasing terms, and the error in truncation has a magnitude smaller than the first neglected term. Thus, for |x| ≤ 1.5, the errors in computing C(x) and S(x) by these power series truncated to N terms are ≤ 2 × 10−16 and ≤ 2.3 × 10−17, respectively, for N = 14.*

*****Ex. 4.4***

**

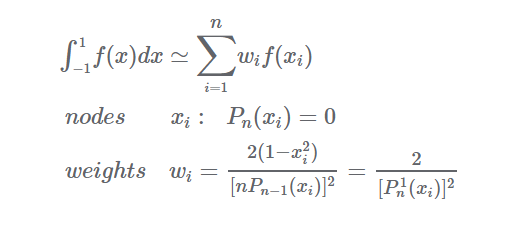
For each n=4,8,16,32,64, the integral will tend to infinity, using the Simpsons rule.

**

Gaussian Rule

import numpy as np  
  
def gauss(f,a,b,n):  
 half = float(b-a)/2.  
 mid = (a+b)/2.  
 [x,w] = np.polynomial.legendre.leggauss(n)  
 result = 0.  
 for i in range(n):  
 result += w[i] \* f(half\*x[i] + mid)  
 result \*= half  
 return result  
  
def fun(x):  
 return np(1/(x-1)(exp(x\*\*2)  
  
for i in range(1,10):   
 print(gauss(fun,1,4,i))

|  |  |  |
| --- | --- | --- |
| *N* | *Value of integral* | *Accuracy* |
| *4* | *2.161971871* | *2.162* |
| *8* | *2.161971871* | *2.161678* |
| *16* | *2.161678001* | *2.161678001* |
| *32* | *2.161678001* | *2.161678001* |
| *64* | *2.161678002* | *2.161678002* |

**

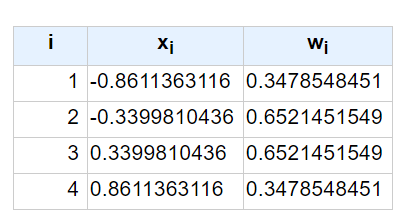
**

Table N=4

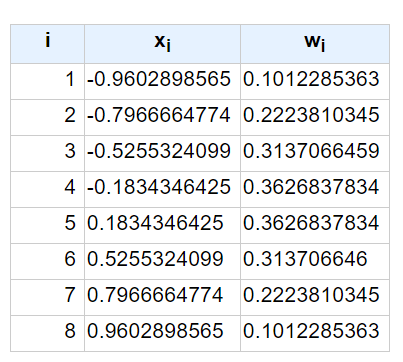
**

Table N=8

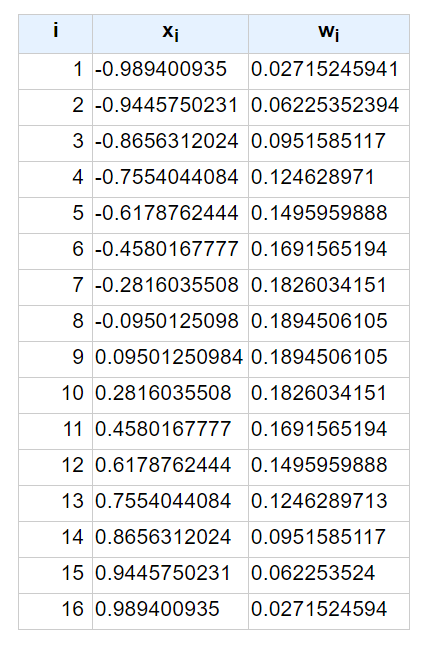
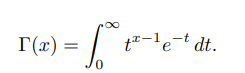
**

Table N=32

***Ex. 4.1***



**c.**

# Python code to demonstrate  
# working of gamma()  
import math  
  
# initializing argument  
gamma\_var = 4  
  
# Printing the gamma value.  
print ("The gamma value of the given argument is : "  
 + str(math.gamma(gamma\_var)))

